Contents:

1 Introduction

Whenever we send data we need to know that what arrives on the other end is identical to what we sent. Similarly, whenever we store data on disk or tape, we need assurance when we retrieve it that it hasn't been altered. Accurate data is absolutely essential for computations, record keeping, transaction processing and online commerce.

Unfortunately, storing and transmitting data is prone to errors - any time while the data being transferred from one place to another can become corrupted in passage. Many factors can alter or wipe out one or more bits of a given data unit, i.e., it could be Single bit Error or Burst Error.

In a ***Single bit Error*** only one bit of a given data unit is changed from 0 to1 or vice versa. A single bit error can be happened if we are sending data using parallel transmission, i.e., if 8 wires are used to send all 8 bits of 1 byte at the same time and one of whose is noisy, one bit can corrupted in each byte. Example: parallel transmission inside CPU and memory.

In a ***Burst Error***, multiple (2 or more) bits are changed, e.g., a 0.01's burst of impulse noise on a transmission with a data rate of 1200bps might change all or some 12 bit of information from 0 to 1 or vice versa. Also, electrical interference (lightning, and other sparks) often produces burst errors.

A system that can't guarantee that the data received by one device are identical to the data transmitted by another device is essentially useless. It is therefore desirable to have some way of detecting --and if possible, correcting-- such errors.

2 Error Detection and Correction

Error detection and correction is the process of detecting errors during the transmission or storage of digital data and correcting them automatically. Error detection is simple than error correction and is the first step in the error correction process.

2.1 Error Detection Mechanisms

Early in the computer revolution, some powerful techniques were developed first to detect errors in data.

2.1.1 Redundancy

Redundancy was the most obvious, and perhaps least efficient, way to find error in received data where the sender repeated each unit of data multiple times. The receiving device would be then be able to do a bit for bit comparison between the copies - any discrepancy would include an error, and an appropriate correction mechanism could be sent in place. This system would be completely accurate, but it would also be insupportably slow. Not only would the transmission time double, but also the time it takes to compare every unit bit by bit must be added. This method is so inefficient that it's not used for error detection -- though the same idea is used in RAID-1 (disk mirroring) for fault tolerance.

2.1.2 Parity

Parity is the best-known error-detection method where a single extra bit is added to each byte of data, called ***parity bit***. There are two types of parity:

- ***Even parity***: Checks if there is even number of ones, if yes then parity bit is zero. When number of one's is odd then parity bit is set to 1.
- ***Odd Parity***: Checks if there is odd number of ones, if yes then parity bit is zero. When number of one's is even then parity bit is set to 1.

Parity checking can be simple or two-dimensional. Parity checking can be simple or two-dimensional.

2.1.2.1 Simple Parity Check

In Simple Parity Check technique a redundant bit called Parity bit, is added to every data unit and assigned a value of 1 or 0, typically according to whether there is an even or odd number of "1" bits.

For example, assume that you are setting a parity bit over all the digits of the word: 1101 0000. Considering '***Even parity***' if the total number of "1s" were even, so the parity bit would be 0. Therefore, the word might then be written as:

1	2	3	4	5	6	7	8	P
1	1	0	1	0	0	0	0	1

Based on this information additional bit is appended to original data. Thus if we consider 8-bit data, then adding parity bit will make it 9 bit. In receiver side, once again parity is calculated and matched with the received parity (bit 9), and if they match data is ok, else data is corrupt. For example, assume that there was an error in the final bit (8th bit):

1	2	3	4	5	6	7	8	P
1	1	0	1	0	0	0	1	0

Parity bit **P** would NOT agree with the transmitted word – which means there must be some error in the 8th bit data.

Parity checking is not very robust, since if more than one bits are changed (burst error) the check bit will be valid and so not detect the error, e.g., error in 5th and 8th bit has occurred:

1	2	3	4	5	6	7	8	P
1	1	0	1	1	0	0	1	1

Parity bit **P** will agree with the parity bit in the transmitted word and thus fail to detect the error.

2.1.2.2 Two Dimensional Parity Check

This technique increases the likely hood of detecting burst error.

2.2 Error Correction Mechanisms

Error correction can be handled in several ways. Two most common are Error Correction by Retransmission and forward error correction.

- ***Error correction by retransmission:*** In this error correction process when a error discovered, the receiver can have the sender transmit the entire data unit.

- ***Forward error correction (FEC):*** in FEC receiver can use an error correction code, which automatically corrects certain errors.

To detect and correct these data errors 'Coding Theory' was originated in 1948 by Richard Hamming. While working on the mechanical relay computer at the Bell-laboratories, Hamming found that much time was lost due to the computer's inability to deal with errors. Hamming began researching and soon came up with a mathematical model for a binary single-error correcting code. The idea was simple and elegant; by cleverly placing parity check digits among the message digits of the code, the series of parity checks will produce the number (in binary) of the position where the error occurred. In 1950 he published what is now known as Hamming Code.

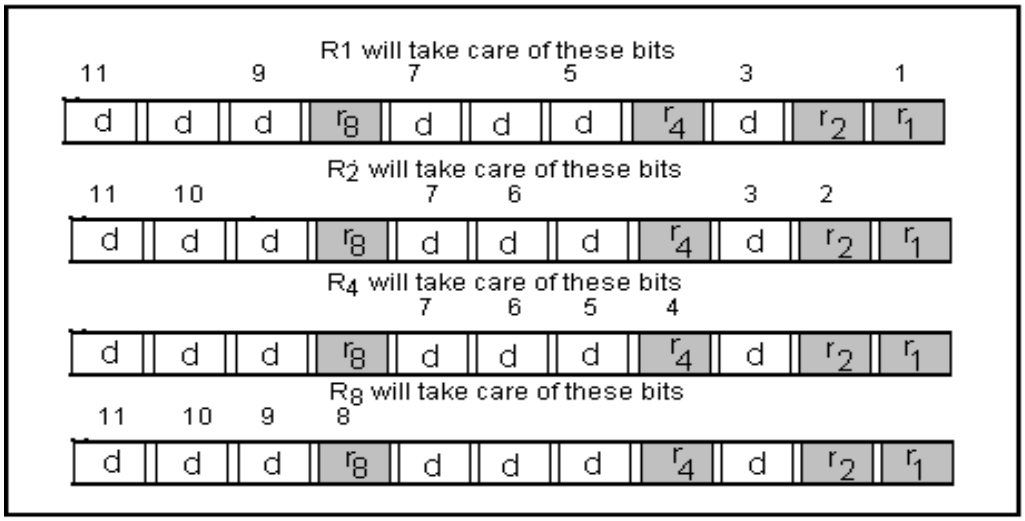
2.2.1 Hamming Code

The Hamming code can be applied to data units of any length and uses the relationship between data and redundancy bits [parity]. For example, a 7-bit ASCII code requires 4 redundancy bits that can be added to the end of the data unit or interspersed with the original data bits. These bits are placed in positions 1, 2, 4 and 8 (the Position in an 11-bits sequence that are powers of 2). For clarity in the examples below, we refer to these bits as r1, r2, r4 and r8.

In Hamming Code, each r bit is parity bit for one combination of data bit

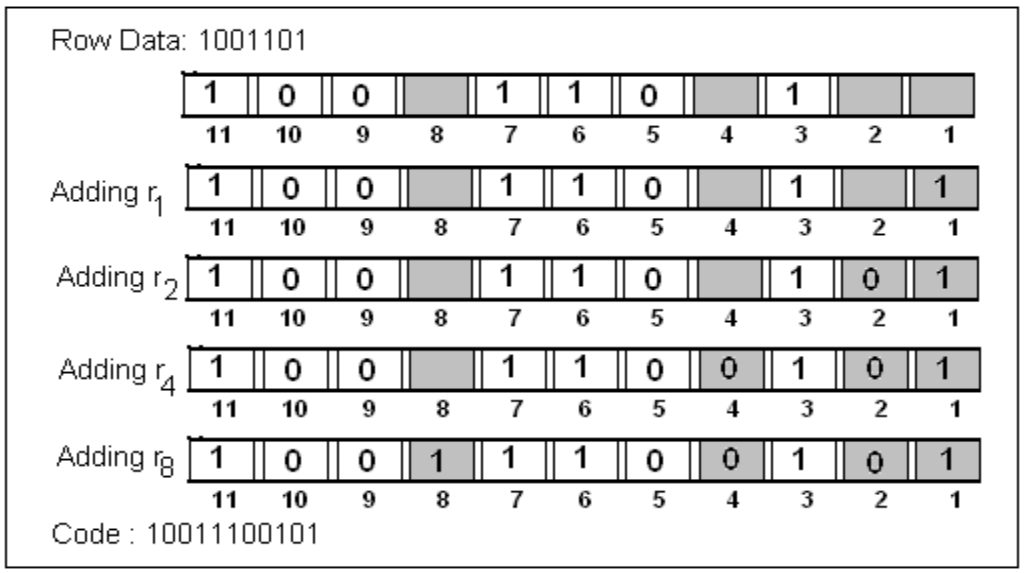
r1: Bits 1,3,5,7,9,11
r2: Bits 2, 3,6,7,10,11
r4: Bits 4,5,6,7
r8: Bits 8, 9, 10, 11

Each data bit may be included in more than one calculation. In the above sequence, i.e., each of the original data bit is included in at last two sets, while the r bits are included in only once.



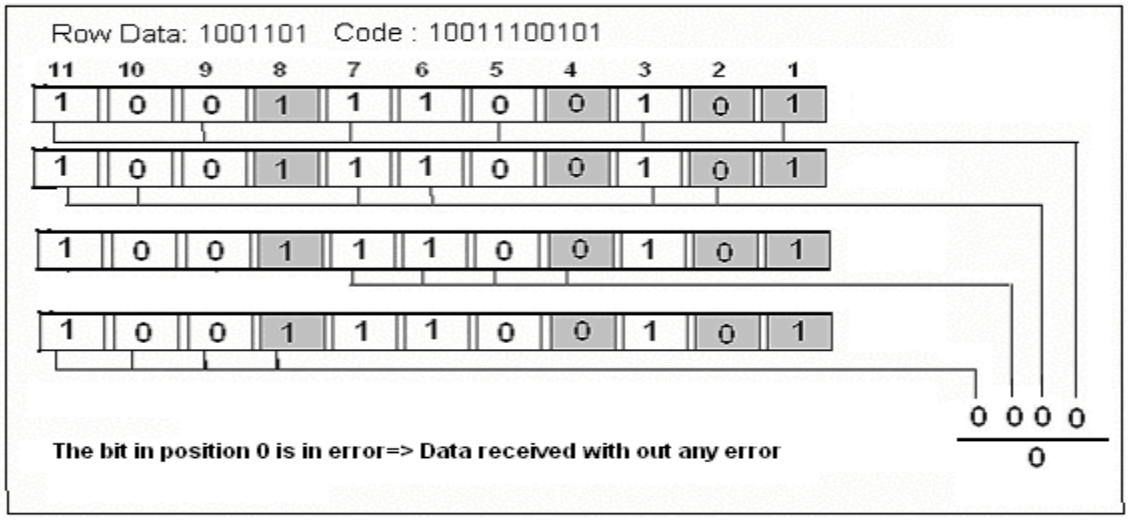
Redundancy bits calculation

The above Figure Shows a Hamming code implementation for ASCII character. In the first step, we place each bit of the original character in its appropriate position in the 11 bit unit. In subsequent steps, we calculate the even parities for the various bit combinations. The parity value for each combination is the value of the corresponding r bit.



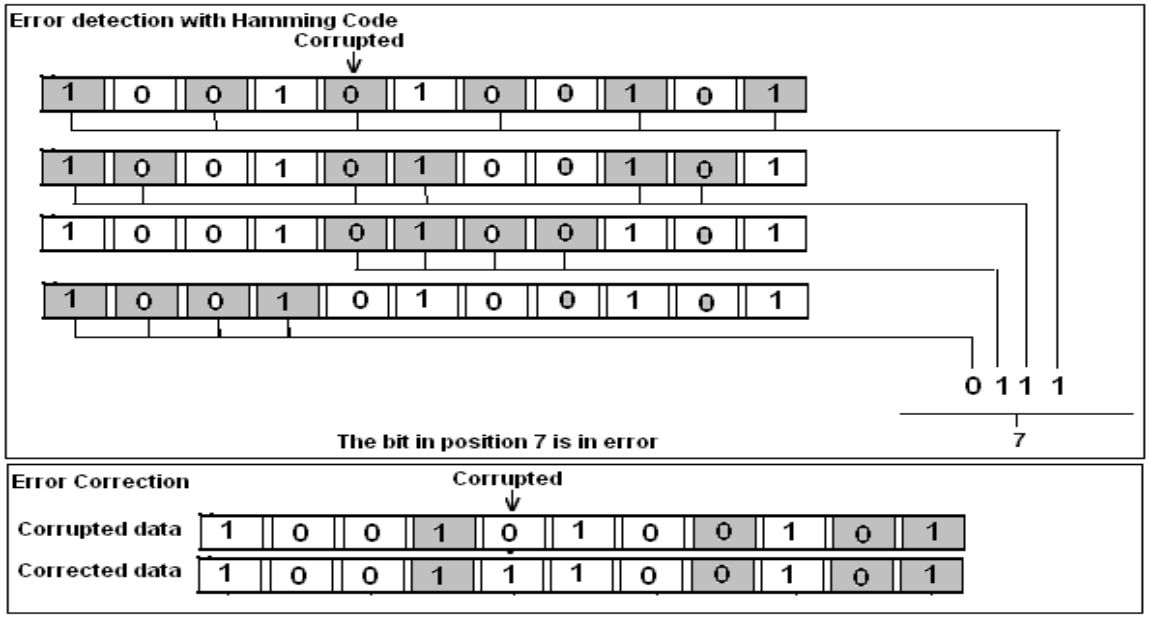
Example of redundancy bit calculation

When the above transmission is received, the receiver takes the transmission & re calculates 4 new parity bits, using the same sets of bits used by the sender plus the relevant parity r bit for each set. Then it assembles the new parity values into a binary number in number of r position (r8, r4, r2, r1). These steps give the precise location of the bit in error. If the number is 0, then it means the data received without any error.



Recalculation of redundancy bit at Receiver end

Now imagine that by the time the above transmission is received, the number 7 bit has been changed from 1 to 0. The receiver takes the transmission & re calculates 4 new parity bits, using the same sets of bits used by the sender plus the relevant parity r bit for each set. Then it assembles the new parity values into a binary number in number of r position (r8, r4, r2, r1).



In the example the steps gives the binary number 0111 (value 7 in decimal), which is the precise location of the bit in error. Once the bit is identified, the receiver can reverse its value and correct the error.

The method described above can correct single bit error only. In theory it is possible to correct any errors automatically. Error correction codes however, are more sophisticated than error detection

3 Implementation

Hamming Code uses the relationship between data and redundancy bits [parity] and can be applied to data units of any length and. In this project I have implemented Hamming Code for a 7-bit ASCII raw data, which requires 4 redundancy bits that interspersed with the original data bits in positions 1, 2, 4 and 8 (the Position in an 11-bits sequence that are powers of 2).

3.1 Platform

Programming Language: C
Operating Systems: Windows

3.2 Uses Guide

Input file (will be transmitted): Some text file named input.txt

Run ECMT: ECMT <Input file>

It convert the characters (from input.txt) ASCII values to its 7bit binary array store it on in7.bin. It then inserts redundancy bit in the array & transfer/store it on rb11.bin

Therefore, output files that will create are

1. in7.bin contain 7bit binary representation of raw data
2. rb11.bin contain 11 bit binary representation of code
   (open with note pad not with textpad)

Run ECM: ECM

Running ECM executable will take rb11.bin (present in the active directory) as input and will create an output file out.txt(output file) containing data in char format (as transmitted).

3.3 Source Code

The transmitter side source is maintained at \src\ECMT.c. **ECMT** executable takes the input file to be transferred and convert the character's (from input file) ASCII values to its 7-bit binary array store it in **in7.bin**. It then inserts redundancy bits in the array & transfer/store it in **rb11.bin**.

The receiver side code is present at \src\ECM.c. **ECM** executable takes received code file **rb11.bin** as input & by taking 11 char from the file at a time it stores them in an integer array. Then check for any error by recalculating its redundancy bits and if any error is there corrects it. It calculate corresponding decimal (ASCII) value by removing RBs from the array and finally retrieves and put the equivalent char in the file **out.txt** (output file)

The header file \src\ECM_common.h contains common utilities shared by both ECMT (Transmitter side) & ECM (Receiver side).

3.3.1 ECM_common.h

```
//-------------------------------------------------------------------------
// File      : ECM_common.h
//
// Author    : Swarna Sekhar Dhar, BCA (Final Semester) Student
//             Dumkal Institute of Engineering & Technology, Murshidabad
//             West Bengal University of Technology (W.B.U.T)
//
// Objective : The Hamming code can be applied to data units of any length &
//             uses the relationship between data & redundancy bits [parity].
//             For example, a 7-bit ASCII code requires 4 redundancy bits
//             that can be added to the end of the data unit or interspersed
//             with the original data bits at the transmitter end before
//             transmission. ECM_common.h contains common utilities shared
//                     by both ECMT (Transmitter side) & ECM (Receiver side).
//
// Assumption: Both ECMT and ECM implemented in following assumption:
//                  Bit wise            : 8 7 6 5 4 3 2 1
//                  Binary Data         : 1 0 0 1 0 0 0 1
//                  C array wise        : 0 1 2 3 4 5 6 7
//-------------------------------------------------------------------------

#ifndef ECM_common_h
#define ECM_common_h

#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<math.h>

// ------------------------------------------------------------------------
// Constant: BIT_LENGTH : 7
// Purpose : binary bit length of raw data
// ------------------------------------------------------------------------

#define BIT_LENGTH 7
```

```
// ------------------------------------------------------------------------
// Constant: REDUNDANCY_BIT_LENGTH : 11
// Purpose : binary bit length of code [raw data  with redundancy bit]
// ------------------------------------------------------------------------

#define REDUNDANCY_BIT_LENGTH 11

// ------------------------------------------------------------------------
// Function: calculate_r1(), calculate_r2(), calculate_r4(), calculate_r8()
// Purpose : In Hamming Code, a 7-bit ASCII code requires 4 redundancy bits
//           that can be added to the end of the data unit or interspersed
//           with the original data bits. These bits are placed in positions
//           1, 2, 4 and 8 (the Position in an 11-bits sequence that are
//           powers of 2). For clarity in the examples below, we refer to
//           these bits as r1, r2, r4 and r8. Each r bit is parity bit for one
//           combination of data bit
//                r1:     Bits 1,3,5,7,9,11
//                r2:     Bits 2, 3,6,7,10,11
//                r4:     Bits 4,5,6,7
//                r8:     Bits 8, 9, 10, 11
//           This function calculate and returns these 'r' values
// Parameters: int ar[] - the 11 bit binary array of code
// Return    : int
// ------------------------------------------------------------------------
int calculate_r1(int ar[])
{
    int r1 = 0;
    // r1 checks 1,3,5,7,9,11 bit
    r1 = ar[REDUNDANCY_BIT_LENGTH - 1] + ar[REDUNDANCY_BIT_LENGTH - 3]
            + ar[REDUNDANCY_BIT_LENGTH - 5] + ar[REDUNDANCY_BIT_LENGTH - 7]
            + ar[REDUNDANCY_BIT_LENGTH - 9] + ar[REDUNDANCY_BIT_LENGTH - 11];
    return (r1 % 2);
}

int calculate_r2(int ar[])
{
    int r2 = 0;
    // r2 chacks 2,3,6,7,10,11
    r2 = ar[REDUNDANCY_BIT_LENGTH - 2] + ar[REDUNDANCY_BIT_LENGTH - 3]
            + ar[REDUNDANCY_BIT_LENGTH - 6] + ar[REDUNDANCY_BIT_LENGTH - 7]
            + ar[REDUNDANCY_BIT_LENGTH - 10] + ar[REDUNDANCY_BIT_LENGTH - 11];
    return (r2 % 2);
}

int calculate_r4(int ar[])
{
    int r4 = 0;
    // r4 chacks 4,5,6,7
    r4 = ar[REDUNDANCY_BIT_LENGTH - 4] + ar[REDUNDANCY_BIT_LENGTH - 5]
            + ar[REDUNDANCY_BIT_LENGTH - 6] + ar[REDUNDANCY_BIT_LENGTH - 7];
    return (r4 % 2);
}

int calculate_r8(int ar[])
{
    int r8 = 0;
    // r8 chacks 8,9,10,11
    r8 = ar[REDUNDANCY_BIT_LENGTH - 8] + ar[REDUNDANCY_BIT_LENGTH - 9]
            + ar[REDUNDANCY_BIT_LENGTH - 10] + ar[REDUNDANCY_BIT_LENGTH - 11];
    return (r8 % 2);
}

#endif /*ECM_common_h*/
```

3.3.2 ECMT.c

```
//-------------------------------------------------------------------------
// File      : ECMT.c
//
// Author    : Swarna Sekhar Dhar, BCA (Final Semester) Student
//             Dumkal Institute of Engineering & Technology, Murshidabad
//             West Bengal University of Technology (W.B.U.T)
//
// Objective : The Hamming code can be applied to data units of any length &
//             uses the relationship between data & redundancy bits [parity].
//             For example, a 7-bit ASCII code requires 4 redundancy bits
//             that can be added to the end of the data unit or interspersed
//             with the original data bits at the transmitter end before
//             transmission. ECMT takes the input file to be transferred and
//                  1. Convert the charecter's (from input file ) ascii values
//                     to its 7 bit binary  array store it in in7.bin
//                  2. Inserts redundancy bits in the array & transfer/store
//                     it in rb11.bin
//
// Uses Guide: 1. Run ECMT executable with the input file
//                  e.g., ECMT in.txt
//             2. Output files that it will create
//                - in7.bin contain 7bit binary represantation
//                - rb11.bin contain 11 bit binary represantation
//                  (view with notepad only)
//-------------------------------------------------------------------------



#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

#include "ECM_common.h"


// For Debugging
#define DEBUG_ECMT  1


// ------------------------------------------------------------------------
// Function     : replaceArray(..)
// Purpose      : This functions replace the raw data binary array by code.
// Parameters   : int code[] - the binary array of code
//                int rawData[] - the binary array of raw data
//                int l - length of binary arrays
// Return       : void
// ------------------------------------------------------------------------

void replaceArray(int code[], int rawData[], int l)
{
    int i = 0;
    for (i = 0; i < l; i++)
    {
        rawData[i] = code[i];
    }
}
```

```
// ------------------------------------------------------------------------
// Function     : insertRedundancyBit(..)
// Purpose      : This functions insert parity bit in the binary array of
//                raw data and thus creates the code.
// Parameters   : int binBuf[] - the binary array of raw data
// Return       : void
// ------------------------------------------------------------------------

void insertRedundancyBit(int binBuf[])
{
    int a = 0, b = 0;
    int riBinBuf[REDUNDANCY_BIT_LENGTH] = {0};

    for (a = 0 ; a < REDUNDANCY_BIT_LENGTH; a++)
    {
        if ((a == REDUNDANCY_BIT_LENGTH - 1)
            || (a == REDUNDANCY_BIT_LENGTH - 2)
            || (a == REDUNDANCY_BIT_LENGTH - 4)
            || (a == REDUNDANCY_BIT_LENGTH - 8))
        {
            riBinBuf[a] = 0;
        }
        else
        {
            riBinBuf[a] = binBuf[b];
            b++;
        }
    }

    // Set the Redundant (parity)  bits
    riBinBuf[REDUNDANCY_BIT_LENGTH - 1] = calculate_r1(riBinBuf); //r1
    riBinBuf[REDUNDANCY_BIT_LENGTH - 2] = calculate_r2(riBinBuf); //r2
    riBinBuf[REDUNDANCY_BIT_LENGTH - 4] = calculate_r4(riBinBuf); //r4
    riBinBuf[REDUNDANCY_BIT_LENGTH - 8] = calculate_r8(riBinBuf); //r8

    replaceArray(riBinBuf, binBuf, REDUNDANCY_BIT_LENGTH);
}



// ------------------------------------------------------------------------
// Function     : main(..)
// Purpose      : This functions main function and entry point for ECMT.
// Parameters   : Takes argument counter and argument vector
// Return       : void
// ------------------------------------------------------------------------

void main(int argc,char* argv[])
{
    char ch = '\0';
    int i = 0, j = 0, l = 0;
    int binaryBuf[REDUNDANCY_BIT_LENGTH] = {0};
    FILE *inputFp = NULL, *outputFp = NULL, *outRbFp = NULL;

    // Commandline Input Format: ECMT <input file name>
    if (argc != 2)
    {
        fprintf(stderr, "Error: Input Format is: ECMT <input file name>\n");
        return;
    }
```

```
    // Open the input file in read mode.
    inputFp = fopen(argv[1], "r");
    if (inputFp == NULL)
    {
        fprintf(stderr, "Error: Unable to Open Input file %s \n", argv[1]);
        return;
    }

    // Open output files:
    //  - in7.bin contain 7 bit raw data binary represantation
    //  - rb11.bin contain 11 bit code binary represantation
    outputFp = fopen("in7.bin","w");
    outRbFp= fopen("rb11.bin","w");

    // Convert each charrecter of the input file to its 7bit binary values
    // and put it in the output binary file "in7.bin".
    do
    {
        ch = getc(inputFp);
        i = toascii(ch);
        l = BIT_LENGTH - 1;

        do
        {
            j = (i % 2);
            binaryBuf[l] = j;
            i /= 2;
            l--;
        } while (l >= 0);

        if (DEBUG_ECMT)
        {
            printf("\nRaw data(%c):", ch);
        }

        for (l = 0; l < BIT_LENGTH; l++)
        {
            if (DEBUG_ECMT)
            {
                printf("%1d",binaryBuf[l]);
            }

            fprintf(outputFp,"%1d",binaryBuf[l]);
        }

        // Insert Redundancy Bit
        insertRedundancyBit(binaryBuf);

        printf("\tCode(11-bit):");

        for (l = 0; l < REDUNDANCY_BIT_LENGTH; l++)
        {
            printf("%1d",binaryBuf[l]);
            fprintf(outRbFp,"%1d",binaryBuf[l]);
        }
    } while (feof(inputFp) == 0);

    // Close all files.
    fclose(inputFp);
    fclose(outputFp);
    fclose(outRbFp);
}
```

3.3.3 ECM.c

```
//-------------------------------------------------------------------------
// File      : ECM.c
//
// Author    : Swarna Sekhar Dhar, BCA (Final Semester) Student
//             Dumkal Institute of Engineering & Technology, Murshidabad
//             West Bengal University of Technology (W.B.U.T)
//
// Objective : The Hamming code can be applied to data units of any length &
//             uses the relationship between data & redundancy bits [parity].
//             For example, a 7-bit ASCII code requires 4 redundancy bits
//             that can be added to the end of the data unit or interspersed
//             with the original data bits at the transmitter end before
//             transmission. ECM takes received code file rb11.bin as input &
//             by taking 11 char from the file at a time
//              1. Store it in int array
//              2. Check for any error by recalculating its redundancy bits
//              3. If any error is there corrects it then calculate
//                 Corresponding decimal (ASCII) value by removing RBs
//              4. Put the equivalent char in the file out.txt (output file)
//
// Uses Guide: 1. Run ECM executable. Here rb11.bin present in the active
//                directory acts as input.  e.g., ECM
//             2. Output files that it will create
//                - out.txt contain characters as transmitted
//-------------------------------------------------------------------------

#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<math.h>

#include "ECM_common.h"

// For Debugging
#define DEBUG_ECM  1

#define NOT(x) ((x == 1) ? 0:1)


// -------------------------------------------------------------------------
// Function     : getCharFromRawData(..)
// Purpose      : This functions retrieves charecter from 7 bit raw data
// Parameters   : int code[] - the 11 bit binary array of code
// Return       : void
// -------------------------------------------------------------------------

char getCharFromRawData(int b7[])
{
       int l = 0, lo = 0, p = 0, r = 0;

       // Convert a 7 bit binary to its equivalent char
       for (l = 0; l < BIT_LENGTH; l++)
       {
              p = (int) pow(2, BIT_LENGTH - l - 1) * b7[l];
              r = lo + p;
              lo = r;
       }
       return((char) lo);
}
```

```
// ------------------------------------------------------------------------
// Function     : removeRedundancyBitAndGetCharFromRawData(..)
// Purpose      : This functions extracts raw data by removing redundancy
//                          bits from code
// Parameters   : int code[] - the 11 bit binary array of code
// Return       : void
// ------------------------------------------------------------------------

char removeRedundancyBitAndGetCharFromRawData(int code[])
{
       int a = 0, b7 = 0;
       int rawData[7] = {0};

       // Get raw data
       for (a = 0; a < REDUNDANCY_BIT_LENGTH; a++)
       {
              if ((a != (REDUNDANCY_BIT_LENGTH - 1))
                     && (a != (REDUNDANCY_BIT_LENGTH - 2))
                     && (a != (REDUNDANCY_BIT_LENGTH - 4))
                     && (a != (REDUNDANCY_BIT_LENGTH - 8)))
              {
                     rawData[b7] = code[a];

                     if (DEBUG_ECM)
                     {
                            printf("%d",rawData[b7]);
                     }
                     b7++;
              }
       }
       return (getCharFromRawData(rawData));
}


// ------------------------------------------------------------------------
// Function     : checkAndCorrectError(..)
// Purpose      : This functions checks for error bit and
//                          make correction in the array if error is there.
// Parameters   : int code[] - the 11 bit binary array of code
// Return       : void
// ------------------------------------------------------------------------

void checkAndCorrectError(int code[])
{
       int errorLoc = 0, errorBit = 0;
       int ch[4] = {0};

       // Recalculate redundancy bit at the receiver end
       ch[0]= calculate_r1(code);
       ch[1]= calculate_r2(code);
       ch[2]= calculate_r4(code);
       ch[3]= calculate_r8(code);

       // Find location of error. This will be zero if no error is there.
       errorLoc = (ch[0] * (int) pow(2,0)) + (ch[1] * (int) pow(2,1))
              + (ch[2] * (int) pow(2,2)) + (ch[3] * (int) pow(2,3));

       if(errorLoc != 0)
       {
              // Get the erroneous bit and correct it
              errorBit = code[REDUNDANCY_BIT_LENGTH - errorLoc];
              code[REDUNDANCY_BIT_LENGTH - errorLoc]= NOT(errorBit);
```

```
                if (DEBUG_ECM)
                {
                        printf(" [ERROR at %d bit]", errorLoc);
                }
                return;
        }

        if (DEBUG_ECM)
        {
                printf(" [No Error Recv.]");
        }
}


// ------------------------------------------------------------------------
// Function     : main(..)
// Purpose      : This functions main function and entry point for ECM.
// Parameters   : None
// Return       : void
// ------------------------------------------------------------------------

void main()
{
        char c = '\0';
        int i = 0, co = 0;
        int b11[REDUNDANCY_BIT_LENGTH] = {0};
        FILE *inputFp = NULL, *outputFp = NULL;

        // Open rb11.bin (contain 11 bit code binary represantation)
        inputFp = fopen("rb11.bin","r");
        if (inputFp == NULL)
        {
                fprintf(stderr, "Error: Unable to Open Input file rb11.bin \n");
                return;
    }

        // Open output file
        outputFp = fopen("out.txt","w");
        if (outputFp == NULL)
        {
                fprintf(stderr, "Error: Unable to Open Output file out.txt \n");
                return;
    }

        while(c=(getc(inputFp)) != EOF)
        {
                co++;
        }

        rewind(inputFp);

        while(co != 0)
        {
                if (DEBUG_ECM)
                {
                        printf("Code (11-bit):");
                }

                // Get code corresponding to a single charecter
                for(i = 0; i < REDUNDANCY_BIT_LENGTH; i++)
                {
                        fscanf(inputFp, "%1d", &b11[i]);
```

```
                        co--;

                        if (DEBUG_ECM)
                        {
                                printf("%d",b11[i]);
                        }
                }

                // Check transmitted data and correct if any error exists
                checkAndCorrectError(b11);

                if (DEBUG_ECM)
                {
                        printf(" Raw data(7-bit):");
                }

                // Get raw data by removing redundancy bits from code
                c = removeRedundancyBitAndGetCharFromRawData(b11);

                if (DEBUG_ECM)
                {
                        printf(" Charecter: %c\n", c);
                }

                fprintf(outputFp,"%c",c);
        }

        // Close all files
        fclose(inputFp);
        fclose(outputFp);
}
```

3.4 Tests

Following are some tests done wit there outputs.

3.4.1 noError_Test

This test depicts an ideal scenario where sender sends encrypted data, which received at the receiver end with out any error.

Input file: input.txt

BCA

Run ECMT

ECMT.exe input.txt > debug.ecmt

Where, debug.ecmt is the console output obtained by enabling DEBUG_ECMT in file ECMT.c

ECMT Outputs:

debug.ecmt
Raw data(B):1000010 Code(11-bit):10010011010 Raw data(C):1000011 Code(11-bit):10010011101 Raw data(A):1000001 Code(11-bit):10010000100 Raw data(ÿ):1111111 Code(11-bit):11111111111
in7.bin
10000101000011100000111111111
rb11.bin
10010011010100100111011001000010011111111111

Now keep generated rb11.bin as is which will depicts an ideal scenario where sender sends encrypted data, which received at the receiver end with out any error when feeded to ECM.

Run ECM

ECM.exe > debug.ecm
Where, debug.ecm is the console output obtained by enabling DEBUG_ECM in file ECM.c

ECM Outputs:

debug.ecm
Code (11-bit):10010011010 [No Error Recv.] Raw data(7-bit):1000010 Charecter: B Code (11-bit):10010011101 [No Error Recv.] Raw data(7-bit):1000011 Charecter: C Code (11-bit):10010000100 [No Error Recv.] Raw data(7-bit):1000001 Charecter: A Code (11-bit):11111111111 [No Error Recv.] Raw data(7-bit):1111111 Charecter: □
Out.txt
BCA□

3.4.2 ErrorAt7thBit_Test

This test depicts a scenario where sender sends encrypted data, which received at the receiver end with error in a data bit.

Input file: input.txt

BCA

Run ECMT

ECMT.exe input.txt > debug.ecmt
Where, debug.ecmt is the console output obtained by enabling DEBUG_ECMT in file ECMT.c

ECMT Outputs:

debug.ecmt
Raw data(B):1000010 Code(11-bit):10010011010 Raw data(C):1000011 Code(11-bit):10010011101 Raw data(A):1000001 Code(11-bit):10010000100 Raw data(ÿ):1111111 Code(11-bit):11111111111
in7.bin
10000101000011100000111111111
rb11.bin
10010011010100100111011001000010011111111111

Insert Error

Now modify generated rb11.bin as it will simulate a scenario where sender sends encrypted data, which received at the receiver end with error at 7th bit of 2nd character.

Modified rb11.bin
10010011010100110111011001000010011111111111

Run ECM

ECM.exe > debug.ecm

Where, debug.ecm is the console output obtained by enabling DEBUG_ECM in file ECM.c

ECM Outputs:

debug.ecm
Code (11-bit):10010011010 [No Error Recv.] Raw data(7-bit):1000010 Charecter: B Code (11-bit):10011011101 [ERROR at 7 bit] Raw data(7-bit):1000011 Charecter: C Code (11-bit):10010000100 [No Error Recv.] Raw data(7-bit):1000001 Charecter: A Code (11-bit):11111111111 [No Error Recv.] Raw data(7-bit):1111111 Charecter: □
Out.txt
BCA□

3.4.3 ErrorAt1stBit_Test

This test depicts a scenario where sender sends encrypted data, which received at the receiver end with error in a redundancy (parity) bit, e.g., r1.

Input file: input.txt

BCA

Run ECMT

ECMT.exe input.txt > debug.ecmt

Where, debug.ecmt is the console output obtained by enabling DEBUG_ECMT in file ECMT.c

ECMT Outputs:

```
debug.ecmt
Raw data(B):1000010 Code(11-bit):10010011010
Raw data(C):1000011 Code(11-bit):10010011101
Raw data(A):1000001 Code(11-bit):10010000100
Raw data(ÿ):1111111 Code(11-bit):11111111111

in7.bin
10000101000011100000111111111

rb11.bin
10010011010100100111011001000010011111111111
```

Insert Error

Now modify generated rb11.bin as it will simulate a scenario where sender sends encrypted data, which received at the receiver end with error at 1st bit of 1st character.

```
Modified rb11.bin
10010011011100100111011001000010011111111111
```

Run ECM

ECM.exe > debug.ecm

Where, debug.ecm is the console output obtained by enabling DEBUG_ECM in file ECM.c

ECM Outputs:

```
debug.ecm
Code (11-bit):10010011011 [ERROR at 1 bit] Raw data(7-bit):1000010
Charecter: B
Code (11-bit):10010011101 [No Error Recv.] Raw data(7-bit):1000011
Charecter: C
Code (11-bit):10010000100 [No Error Recv.] Raw data(7-bit):1000001
Charecter: A
Code (11-bit):11111111111 [No Error Recv.] Raw data(7-bit):1111111
Charecter:

Out.txt
BCA
```